Lazy Abstractionと精練を用いた組込みアセンブリプログラムのリアルタイム性のソフトウェアモデル検査 (アルゴリズムと計算理論の新潮流)

AUTHOR(S):

上出, 浩夢; 山根, 智

CITATION:

上出, 浩夢 ...[et al]. Lazy Abstractionと精練を用いた組込みアセンブリプログラムのリアルタイム性のソフトウェアモデル検査 (アルゴリズムと計算理論の新潮流). 数理解析研究所講究録 2020, 2154: 1-8

ISSUE DATE:

2020-04

URL:

http://hdl.handle.net/2433/255105

Lazy Abstractionと精練を用いた組込みアセンブリプログラムのリアルタイム性のソフトウェアモデル検査

上出　浩夢　　山根　智

金沢大学大学院自然科学研究科

Hiromu Kamide　　Satoshi Yamane

Graduate School of Natural Science and Technology, Kanazawa University

1　研究背景・目的

近年，組込みシステムは自動車やデジタル家電など様々な製品に用いられており，複雑化，大規模化している．そのため，システムの安全性を素早く自動で行えることが求められ，その手法として，モデル検査 [1] と呼ばれる形式的手法を用いた検証方法がある．しかし，大規模なモデル検査では状態爆発により，実用的な時間内にモデル検査が終わらないといった問題が発生する．そこで，モデルの状態数を減らすための手法として抽象化の技術が重要となる．また，組込みシステムではシステムの論理的妥当性だけでなくリアルタイム性の保証も重要である．リアルタイム性とは，ある処理に入力を与えた場合決められた時間以内に出力を返すといった性質である．

本論文では，組込みアセンブリプログラムに対し，Lazy Abstraction と呼ばれる抽象化手法と Interpolants による精練を用いて状態数を削減し，リアルタイム性も考慮したモデル検査手法を提案する．

2　モデル検査

モデル検査は形式的検証手法の一つであり，対象となるシステムやプログラムを数学的に等価な状態モデルに変換し，その状態モデル上を網羅的に探索することによって求めたい性質を満たすか検証する．検証したいシステム構造のモデルを M，M 上の状態を s，満たすべき性質を ϕ とすると

$$M, s \models \phi \tag{1}$$

を全ての状態に対して検証することで，システムが性質を満たすかどうか検証する．

2.1 Lazy Abstraction

モデルの一部分に必要な述語のみを追加することで状態数の削減を行う Lazy Abstraction[2] と呼ばれる手法がある．Lazy Abstraction は前方エラー探索と後方反例解析による 2 つのフェーズからなる．

前方エラー探索では，モデルの制御フローオートマトン (CFA:Control Flow Automaton) を深さ優先で展開していき，抽象到達可能木 (ART:Abstract Reachability Tree) を作成していく．この ART 上で検証性質を満たすかモデル検査を行う．もし検証性質を満たさない場合，後方反例解析に進む．

後方反例解析では，性質を満たさないパスが実反例か偽反例かを判断し，実反例ならモデルは性質を満たさないとして検証は終了する．偽反例なら抽象度を下げるため，ART を精練するための抽象述語を求める．この抽象述語を用いて，ART を精練して再び前方エラー探索を行う．

2.2 Interpolation

Lazy Abstraction を行う際に，効率的に反例解析を行うための手法として，Craig Interpolation を用いた手法 [3] がある．これは，プログラムパスの反駁から導出された補間を使用して抽象モデルを精練していく手法である．これにより，抽象遷移関係の計算を計算するコストがかからず，大幅なパフォーマンス向上が得られる．

本稿では，プログラムをモデル化するために，一階述語論理 (FOL) 式を用いる．ここで S を状態集合，$L(S)$ を=や+など様々な解釈された記号を含む状態式，$L(S \cup S')$ を遷移式とする．

この時，プログラムを以下で定義する．

$$P = (\Lambda, \Delta, l_i, l_f) \tag{2}$$

ここで，Λ はプログラムロケーションの集合，Δ は行動の集合，$li \in \Lambda$ は初期ロケーションの集合，$lf \in \Lambda$ エラー状態の集合である．

行動は以下で定義する．

$$\Delta = (l, T, m) \tag{3}$$

ここで，l は行動の開始位置，T は状態遷移式，m は行動の終了位置である．

また，プログラムのパス π は，連続式 $(l_0, T_0, l_1)(l_1, T_1, l_2)...(l_{n-1}, T_{n-1}, l_n)$ で表される．$l_0 = l_i, l_n = l_f$ のとき，パスはエラーパスに到達する．

Interpolation は一般的に，定理などとセットで用いられることが多く，補間定理のことを指す．本研究では以下のクレイグの補間定理 (Craig's interpolation theorem) を用いる．

2 つの式 (A, B) が与えられ，$A \wedge B$ の一貫性がなく，以下を満たすとき，$\hat{A}$ を (A, B) の補間という．

- $A \rightarrow \hat{A}$
- $\hat{A} \wedge B$ が充足不能
- $\hat{A} \in L(A) \cap L(B)$

クレイグの補間の補題によると，一貫性のない FOL 式では，常に補間が存在すると述べている．プログラムのパスを扱うために，この考えを式に一般化する．連続式 $\Gamma = A_1,...,A_n$ が与えられ，以下を満たすとき，$\hat{A}_0,...,\hat{A}_n$ を Γ の補間という．

- $\hat{A}_0 = TRUE$ かつ $\hat{A}_n = FALSE$
- 全ての $1 \leq i \leq n$ について，$\hat{A}_{i-1} \wedge A_i \rightarrow \hat{A}_i$
- 全ての $1 \leq i \leq n$ について，$\hat{A}_i \in (L(A_1...A_i)) \cap (L(A_{i+1}...A_n))$

つまり，補間の i 番目の要素は，共通語彙の接頭辞 $A_0...A_i$ と接尾辞 $A_{i+1}...A_n$ に対する式であり，各補間は A_i と共に次を暗示する．もし Γ が量化子を持たなければ，Γ の反駁から Γ のための量化子を持たない補間を導出できる．

2.3 アセンブリプログラムのモデル化

実際のマイコンをモデル化する事例として，H8/3687 のマイコン[1]を用いる．このマイコン及びアセンブリプログラムのモデル化に関しては [4] を参考とした．

マイコンの構成要素として汎用レジスタ (16bit × 16 本)，プログラムカウンタレジスタ (PC, 24bit × 1 本)，条件コードレジスタ (CCR, 8bit × 1 本)，アドレス空間 (通常モードで 64kByte) を持つ．これらを固定長ビットベクトルの背景理論に基づく型も持つとみなし，モデル化を行う．後方反例解析において自動で補間を求めるために，アセンブリプログラムの実行命令を SMT-LIB[2]の入力形式及び静的単一代入 (SSA：Static Single Assignment) 形式へ変換する必要がある．SMT-LIB は後述する SMTsolver の標準入力形式であり，SSA 形式は各変数が一度のみ代入されるよう定義されたものである．式 4 の命令における SMT-LIB 形式を式 5 であり，これを SSA 形式に変換した述語論理式が 6 である．6 では，PC や CCR に関する暗黙的な振る舞いを省略し，命令の振る舞いを示す論理式のみを記載している. 実際には PC や CCR に関する振る舞いも論理式化し，命令に対する振る舞いを示す論理式との連言として構成する．

$$MOV.L\ ERs,\ ERd \tag{4}$$

$$(=\ \ ERd\ \ ERs) \tag{5}$$

$$(=\ \ ERd_{i+1}\ \ ERs_i) \tag{6}$$

3 提案手法

本研究では，リアルタイム性の検証のためにアセンブリプログラムを検証対象とし，抽象化を導入することで状態削減を行う．そのために，アセンブリプログラムに対する時間付きのモデル化及び Interpolation を用いた抽象モデルの精練手法を提案する．全体的な検証の流れは図 1 の通りである．

[1] https://www.renesas.com/jp/ja/doc/products/mpumcu/002/rjj09b0151_h83687.pdf
[2] http://smtlib.cs.uiowa.edu/

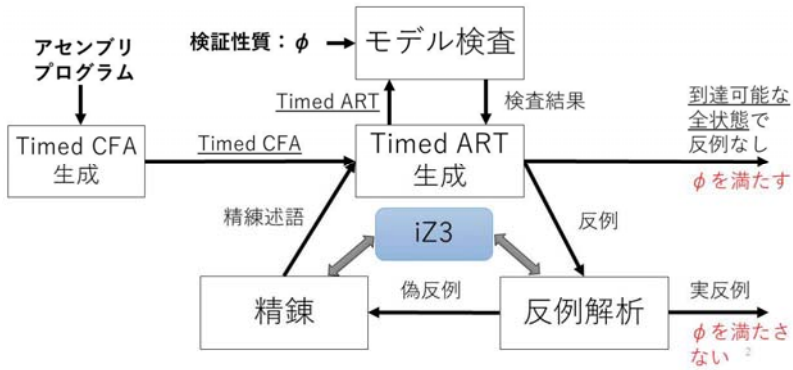

図 1: モデル検査全体の流れ

3.1 時間付きモデル

Timed CFA を以下に定義する

$$C = (V, E, M_v, M_e, T) \tag{7}$$

ここで，(V,E) は有向木の頂点と辺，$M_v : V \to \Gamma$ はプログラムのロケーションへのマップ，$M_e : E \to \Delta$ はプログラムの行動のマップ，$T : E \to time$ はプログラムの行動から命令実行時間へのマップである．

C を用いて Timed ART を以下に定義する

$$A = (C, \psi, \triangleright) \tag{8}$$

ここで，$\psi : V \to L(S)$ は，一階述語論理式へのラベリング，$\triangleright \subseteq V \times V$ は抽象状態のカバー関係である．

この Timed ART の各状態が初期状態からの命令の実行時間を保持しており，制約時間を超える場合リアルタイム性を満たさないといえる．

3.2 アセンブリプログラムの精練

あるパス π において，検証条件式 φ は以下のようになる．

$$\varphi = \bigwedge_{i=0}^{n} (L(T_i) \wedge L(G_i)) \tag{9}$$

ここで，L は状態遷移式を SMT-LIB 形式及び SSA 形式へラベリングする関数で，G は分岐命令における遷移条件式である．

本研究では iZ3[1] という SMTsolver である Z3 の機能を用いて反例解析及び精練述語の導出を行う．π が実行可能であれば充足可能 (SAT) を返し，精練述語は返さない．π が実行不可能であれば充足不能 (UNSAT) として，各状態で成り立つべき述語の集合を返す．

[1] https://rise4fun.com/iZ3/tutorial/guide

3.3 事例を用いた抽象化及び精練

事例を用いて，アルゴリズムの流れを説明する．検証するアセンブリプログラムは 1 を用いる．

プログラム 1: 検証対象のプログラム

```
1     _sample1:
2         PUSH.W R6 ; 1
3         MOV.W R7, R6 ; 2
4         PUSH.W R5 ; 3
5         MOV.W R0, R5 ; 4
6         MOV.W R5, R5 ; 5
7         BGE L49:8 ; 6
8         MOV.W R5, R0 ; 7
9         SUB.W #0, R0 ; 8
10        MOV.W R0, R5 ; 9
11    L49:
12        MOV.W R5, R5 ; 10
13        BGE L50:8 ; 11
14        _error
15        BRA L51 ; 12
16    L50:
17        MOV.W R5, R0 ; 13
18        BRA L52:8 ; 14
19    L51:
20    L52:
21        POP.W R5 ; 15
22        POP.W R6 ; 16
23        RTS ; 17
```

このプログラムは，レジスタ R0 の絶対値を求めるプログラムである．R0 の値が正のとき 7 行目の BGE 命令でラベル 49 に分岐し，負のとき 9 行目の SUB 命令で符号を反転させる．ここで，14 行目の _error をエラー状態とする．13 行目の BGE 命令では，常に値が正であるので実際にこのプログラムを動作させる場合，14 行目には到達しない．抽象化を用いたモデル検査を行う場合この 14 行目に到達する可能性があり，そのパスを精練することでエラー状態に到達しないことを検証する．

プログラム 1 に対して Timed CFA を生成すると図 2 のとおりになる．ここで，CFA の各ノード番号は，プログラム 1 の ‘ ; ’ 後の各番号に対応する．

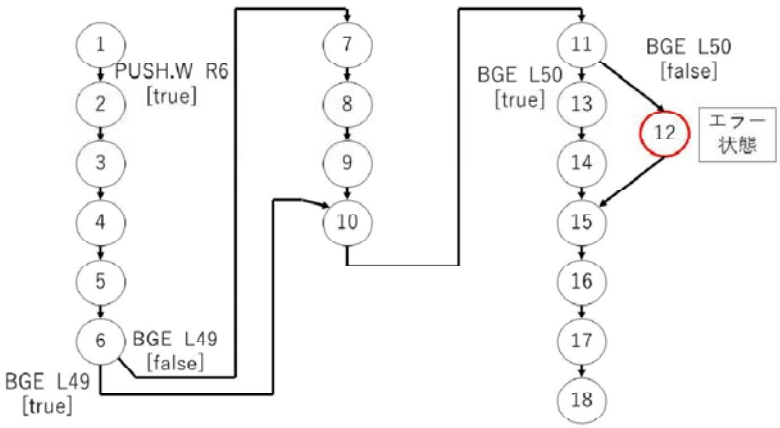

図 2: CFA

図 2 を元に Timed ART を作成し，検証性質を満たすかモデル検査する．エラー状態にたどり着くまで続けると図 3 になる．図 3 では，述語なし (true) で抽象化された抽象状態を $1'$ から順に作成している．初めに CFA の BGE 命令で，状態 6 から 10, 状態 11 から 13 へ遷移するパスを考える．このパスを元に Timed ART を作成すると $1' \rightarrow 2' ... \rightarrow 6' \rightarrow 11' \rightarrow 13' \rightarrow 14' \rightarrow ... \rightarrow 18'$ となる．この時，エラー状態に到達せずにプログラムの終了までたどり着くので，他の遷移先でもエラー状態に遷移するか試す．

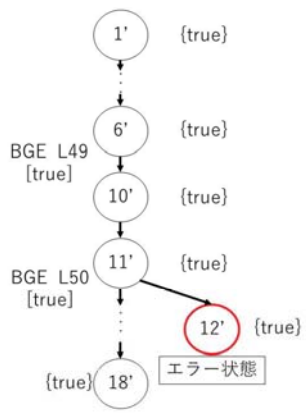

図 3: 前方エラー探索で作成される Timed ART

次に，BGE 命令で状態 6 から 10 に遷移し，状態 11 からで 12 に遷移するパスを考える．このパスを元に Timed ART を作成すると $1' \rightarrow 2' ... \rightarrow 6' \rightarrow 11' \rightarrow 12'$ となる．この時，エラー状態に到達するのでパスを SMT-LIB 形式及び SSA 形式に変換して，iZ3 で反例解析を行う．反例解析の結果，図 3 の $11' \rightarrow 12'$ は偽反例となるため，精練述語が得られ状態 $12'$ には false がラベル付けされると考えられる．

4 実験

本研究では，提案手法で示した事例のプログラムにおいて，エラー状態に到達した場合どのような反例が得られ，Timed ART が精練されるかを検証した．得られた反例が図 4 である．これを元に Timed ART を精練した結果が図 5 である．この結果から，エラー状態には false がラベル付けされ，到達しないことが示された．ここで，状態 6’ から 10’ への遷移では $ER5 \geq 0$ が成り立つため,「最上位ビット (符号ビット) が 0」という述語が割り当てられる．

```
unsat
true
true
true
true
true
(not ((_ bit2bool 31) ER5_0_6))
(not ((_ bit2bool 31) ER5_0_7))
false
```

図 4: iZ3 による Interpolants

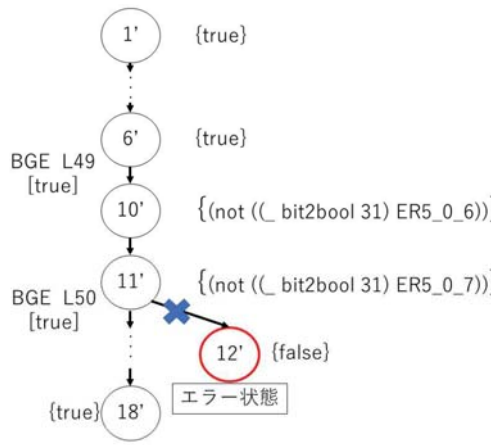

図 5: 精練後の Timed ART

5 まとめ

本研究ではアセンブリプログラムを対象とし，リアルタイム性検証のための時間付きモデル化及び Lazy Abstraction による抽象化と Interpolation を用いた精練による状態削減手法を提案し，実験を行うことで有効性を示した．

今後の研究課題としては，ブロック化によるさらなる状態削減がある．現在 Timed ART は 1 つのノードに対して 1 つの命令しか持っていないが，遷移先が 1 つである複数のノードをブロック化することで，さらに状態を削減可能であると考えられる．

謝辞

This work was supported by the Research Institute for Mathematical Sciences, an International Joint Usage/Research Center located in Kyoto University.

参考文献

[1] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[2] Ranjit Jhala. *Program verification by lazy abstraction*. University of California, Berkeley, 2004.

[3] Kenneth L McMillan. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*, pp. 123–136. Springer, 2006.

[4] Jumpei Kobashi, Satoshi Yamane, and Atsushi Takeshita. Development of smt-based bounded model checker for embedded assembly program. In *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, pp. 696–698. IEEE, 2014.